Compilers

Dr. Sherin ElGokhy

Lecture#5

Ambiguity
Error Handling

Outline

- Ambiguity
- Extensions of CFG for parsing
  - Precedence declarations
  - Error handling

Ambiguity

- Grammar

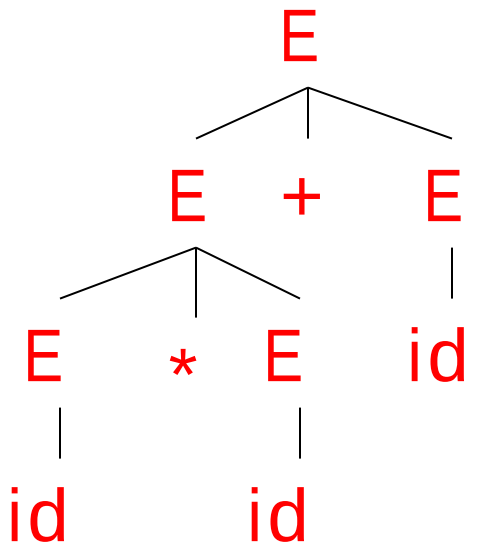
$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

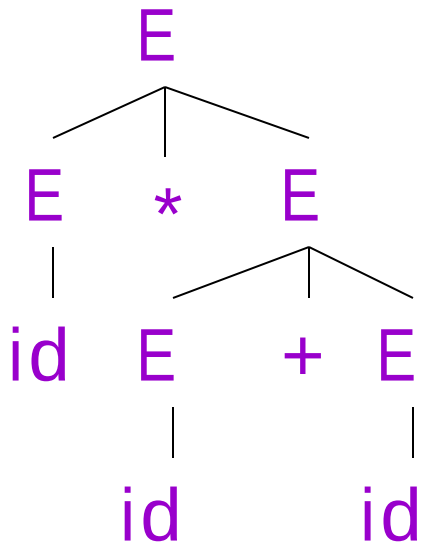
- String

$$id * id + id$$

Ambiguity (Cont.)

This string has two different parse trees for the same string





Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string……distinct parse trees
  - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined
  - Leaves decisions about what the program means to the compiler

Quiz

Which of the following grammars are ambiguous?

- ☐ $S \rightarrow SS \mid a \mid b$
- ☐ $E \rightarrow E + E \mid id$
- ☐ $S \rightarrow Sa \mid Sb$
- ☐ $E \rightarrow E' \mid E' + E$

  $E' \rightarrow -E' \mid id \mid (E)$

Dealing with Ambiguity

- There are several ways to handle ambiguity from a grammer
- Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E' + E \mid E'$$

$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

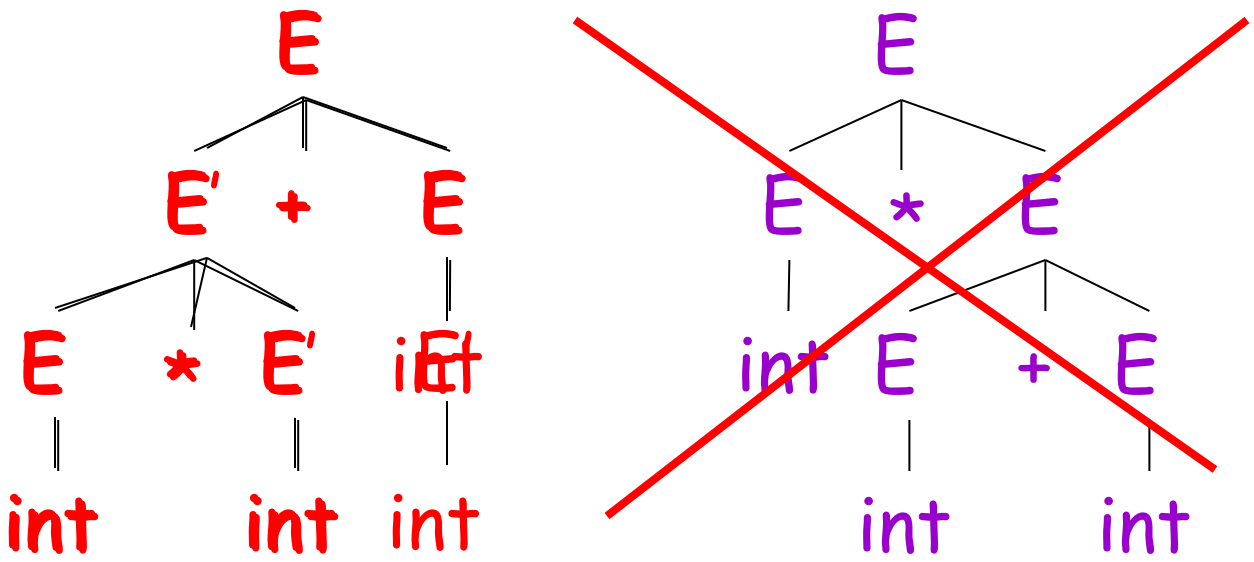
- Enforces precedence of * over + (priority)

Ambiguity in Arithmetic Expressions

- Recall the grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

- The string int * int + int has two parse trees:



Ambiguity: The Dangling Else

- Consider the grammar

  E → if E then E
  | if E then E else E
  | OTHER

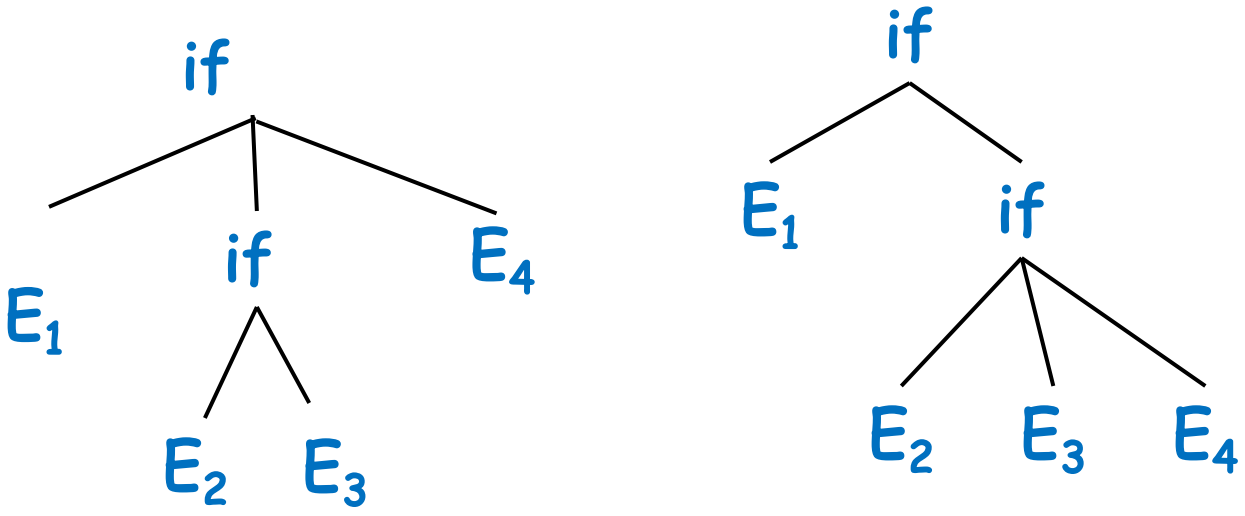
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

- else matches the closest then
- We can describe this in the grammar

E → MIF /* all then are matched with an else */
| UIF /* some then is unmatched */

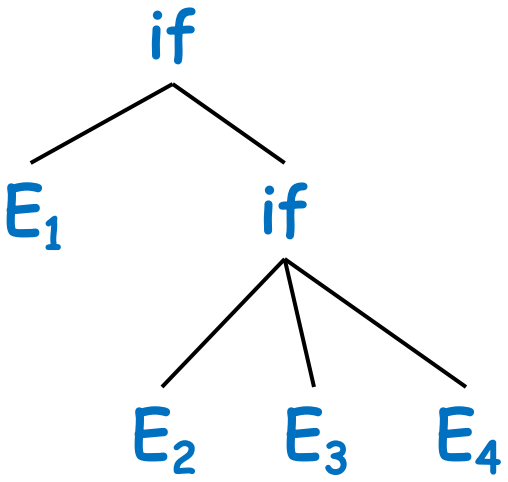
MIF → if E then MIF else MIF
| OTHER

UIF → if E then E
| if E then MIF else UIF

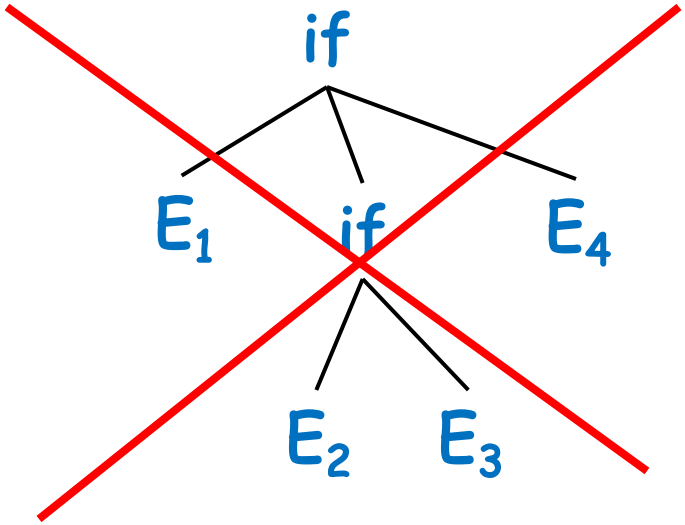
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4



- A valid parse tree (for a **UIF**)



- Not valid because the **then** expression is not a **MIF**

Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Precedence and Associativity Declarations

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars
- Examples …

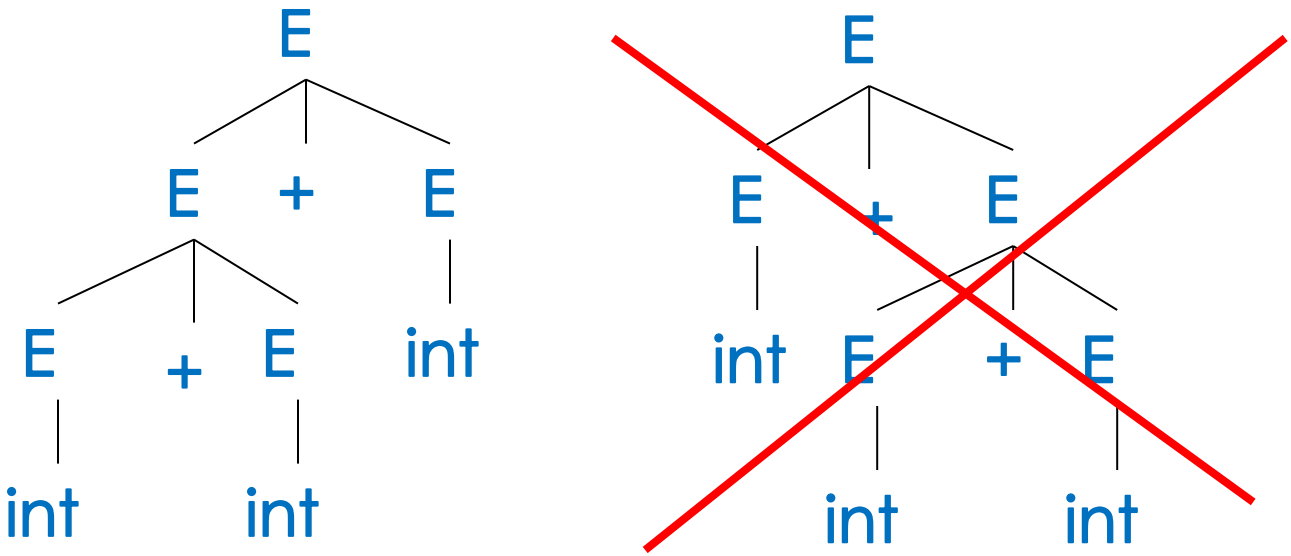
Ambiguity Quiz

Choose the unambiguous version of the given ambiguous grammar: $S \rightarrow SS \mid a \mid b$

- $S \rightarrow S'a \mid S'b$, $S' \rightarrow S \mid \varepsilon$
- $S \rightarrow SS'$, $S' \rightarrow a \mid b$
- $S \rightarrow S \mid S'$, $S' \rightarrow a \mid b$
- $S \rightarrow Sa \mid Sb$

Associativity Declarations

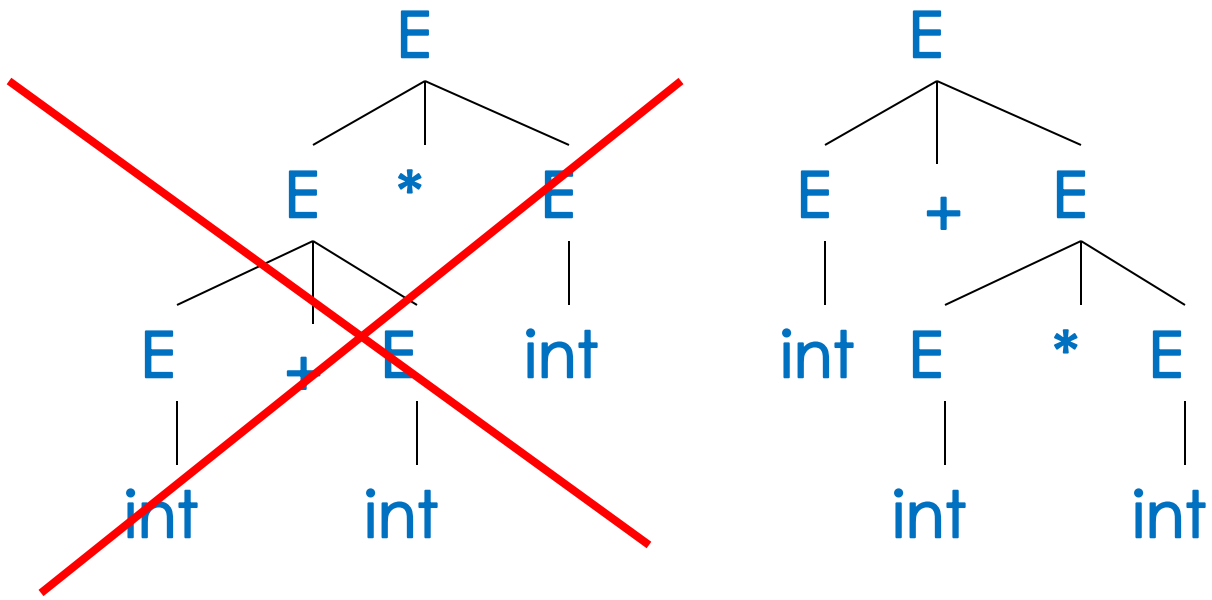
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



- Left associativity declaration: %left +

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int



- Precedence declarations: %left +
  %left *

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	*Example*	*Detected by …*
Lexical	*… $ …*	*Lexer*
Syntax	*… x *% …*	*Parser*
Semantic	*… int x; y = x(3); …*	*Type checker*
Correctness	*your favorite program*	*Tester/User*

Syntax Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code
- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction
- Not all are supported by all parser generators

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there
- Such tokens are called synchronizing tokens
  - Typically the statement or expression terminators

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

- Bison: use the special terminal error to describe how much input to skip

  E → int | E + E | (E) | error int | (error)

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write 5 x instead of 5 * x
  - Add the production E → … | E E
- Disadvantage
  - Complicates the grammar

Error Recovery: Local and Global Correction

- Idea: find a correct "nearby" program
  - Try token insertions and deletions
  - Exhaustive search

- Disadvantages:
  - Hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program
  - Not all tools support it

Syntax Error Recovery: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible
  - Researchers could not let go of the topic
- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
  - Panic-mode seems enough

Thanks